KTH Computer Science and Communication

Storskalig multi-touch

Design och implementation av ett ramverk för gestigenkänning till Microsoft Surface 2.

ANTON HOLMBERG

Kanditatexamenrapport vid NADA
Handledare: Alexander Baltatzis
Examinator: Mårten Björkman

Referat

På senare år har datorer med stöd för användarinput med hjälp av flera fingrar på samma gång på en bildskärm blivit allt vanligare. Microsoft Surface 2 är ett exempel som har en 40 tum stor touchskärm med stöd för över 50 fingrar samtidigt.

I denna rapport undersöks hur man kan implementera ett flexibelt ramverk för interaktion med gestigenkänning på 2D och 3D objekt för Microsoft Surface 2-plattformen. De typer av gester som stöds på objekten är tap, drag, flick, pinch, spread och rotation. Dessutom kan det lätt utvidgas för att stödja nya gester.

Resultatet visar att ramverket ger intuitiv interaktion och bra prestanda med över 50 fingrar på skärmen samtidigt.

Abstract

Large scale multitouch

Computers which support user input on a display with multiple fingers have become increasingly common in recent years. One example is Microsoft Surface 2 with a 40 inch touch display which supports over 50 fingers simultaneously.

This thesis examines how you can implement a flexible framework for gesture recognition on 2D and 3D objects for the Microsoft Surface 2 platform.The supported gestures are tap, drag, flick, pinch, spread and rotation. The framework can also easily be extended to support new gestures.

The results imply that the framework provides intuitive interaction and good performance even when over 50 fingers are present on the screen simultaneously.

Innehåll

1 Introduktion **1**
1.1 Syfte . . . 1
1.2 Problemformulering . . . 2

2 Bakgrund **3**
2.1 Fingergester . . . 3
2.2 Microsoft Surface 2 . . . 6
2.3 Tidigare arbeten . . . 6

3 Ramverksarkitektur **7**
3.1 Överblick . . . 7
3.2 Komponenter . . . 8
3.3 Gestigenkännare . . . 9

4 Algoritmer **11**
4.1 Tap . . . 11
4.2 Flick . . . 11
4.3 Manipulation . . . 12

5 Resultat **13**
5.1 Implementationssvårigheter . . . 13
5.2 Testning . . . 13
5.3 Slutversionen . . . 14

6 Diskussion och slutsats **15**

Litteraturförteckning **17**

Bilagor **18**

A Källkod **19**
A.1 GestureDetector.cs . . . 19
A.2 TapDetector.cs . . . 19
A.3 FlickDetector.cs . . . 21

A.4 ManipulationDetector.cs 23

Kapitel 1

Introduktion

På senare år har datorer med stöd för användarinput med hjälp av flera fingrar på samma gång på en bildskärm blivit allt vanligare, vilket kallas multi-touch. Allt från mobiltelefoner till väggstora displayer utrustas med denna typ av teknologi. Det ger användarna en helt ny sätt att interagera med datorn och nya typer av enheter har börjat etablera sig som endast använder sig av denna typ av interaktion.

Ett helt nytt interaktionsspråk har utvecklats för att beskriva de olika gester som användaren kan göra med sina fingrar på skärmen, t.ex tap, drag, flick, pinch, spread m.m. Vad dessa gester innebär beskrivs i detalj under sektion 2.1.

Det finns ofta färdiga bibliotek för gestigenkänning som programmerare kan använda sig av när man utvecklar till enheter med multi-touch. Men det finns situationer när dessa bibliotek inte kan användas av olika skäl. Ett exempel är om man använder sig av 3D-rendering med ramverk som OpenGL[1] eller XNA[2]. De inbyggda gestigenkänningssystemen bygger ofta på ett färdigt grafiskt ramverk som hanterar 2D-objekt vilket gör att det inte går att använda i detta fall.

1.1 Syfte

Syftet med denna rapport är att undersöka hur man kan utveckla ett ramverk för gestigenkänning på godtyckliga objekt med hög flexibilitet. Det innebär att så lite som möjligt tas för givet kring utvecklingsmiljön där ramverket kommer att användas och att det lätt kan utvidgas. Behovet av detta motiveras av att de flesta ramverk som finns är tätt ihopkopplade med ett färdigt grafikramverk. Exempel på detta är UIKit till iPhone[3] och WPF till Microsoft Surface 2[4] som båda har inbyggt stöd för gestigenkänning. Men oftast går inte att bara använda sig av den delen i kombination med något annan grafikramverk på ett tillfredsställande sätt.

Dessutom undersöks hur man kan implementera gestigenkänning på ett resurssnålt och effektivt sätt med stöd för över 50 fingrar.

1.2 Problemformulering

Den huvudsakliga fråga som denna rapport försöker besvara är:

- Hur kan man utveckla ett flexibelt ramverk för gestigenkänning som klarar av att över 50 fingrar interagerar med flertal objekt samtidigt?

För att kunna svara på den måste ett flertal mindre frågor besvaras:

- Vilken typ av input ska man kräva av systemet där ramverket körs?
- Hur modellerar man interaktiva visuella objekt på ett flexibelt sätt?
- Vilka algoritmer kan man använda för att göra effektiv gestigenkänning?

Kapitel 2

Bakgrund

I detta kapitel beskrivs de vanligaste gesterna som används vid multi-touch i detalj. Sedan ges en introduktion till Microsoft Surface 2-plattformen, både ur ett användar och utvecklarperspektiv. Slutligen tas tidigare arbeten kring multi-touch-ramverk upp.

2.1 Fingergester

Ett helt nytt interaktionsspråk har utvecklats för att beskriva de olika gester som användaren kan göra med sina fingrar vid multi-touch. Det finns ingen standard för vad dessa ska kallas på svenska. I denna rapport kommer följande namn att användas.

Tap

Liknar ett klick med en datormus men man använder fingret istället. Man trycker ett finger mot skärmen och tar sedan snabbt upp det igen. Motsvarigheten till dubbelklick med en mus kallas i denna rapport för dubbeltap. Se figur 2.1.

Figur 2.1. Illustration av gesten *Tap*.

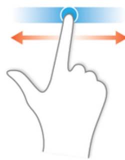
Drag

En gest för att flytta objekt genom att först ha sitt finger på objektet. När man sedan flyttar på fingret följer objektet efter. Se figur 2.2.

Figur 2.2. Illustration av gesten *Drag*.

Flick

En gest där man drar sitt finger snabbt över skärmen i en rak linje i godtycklig riktning. Se figur 2.3.



Figur 2.3. Illustration av gesten *Flick*.

Pinch

En gest där man använder flera fingrar som man drar ihop på ett objekt. Det resulterar i att objektet blir mindre. Se figur 2.4.

Spread

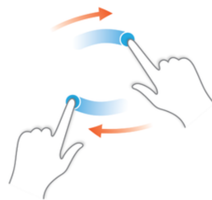
En gest där man använder flera fingrar som man drar ifrån varandra på ett objekt. Det resulterar i att objektet blir större. Se figur 2.5.

Rotation

En gest där man håller in flera fingrar på ett objekt. Sedan snurrar man på fingrarna och objektet snurrar med. Se figur 2.6.

Figur 2.4. Illustration av gesten *Pinch*.

Figur 2.5. Illustration av gesten *Spread*.



Figur 2.6. Illustration av gesten *Rotation*.

2.2 Microsoft Surface 2

Gestigenkänningssystemet ska implementeras på Microsoft Surface 2[5]. Microsoft Surface 2 är en pekskärmsdator utvecklad av Microsoft. Den har en 40 tums skärm och stöd för obegränsat antal fingrar (så många som får plats). Den har redan ett inbyggt gestigenkänningssystem men man kan även få tillgång till den råa inputdata skärmen får. En hittills unik teknologi för plattformen är att det finns ett inbyggt system för att känna igen fysiska objekt på skärmen med hjälp av speciella taggar.

När man utvecklar applikationer till Microsoft Surface 2 finns det två vanliga tillvägagångssätt. Det första är att använda sig av *Microsoft Windows Presentation Foundation* (WPF). Det är Microsofts egna ramverk för utveckling av surfaceapplikationer som inkluderar många färdiga grafiska komponenter och inbyggt stöd för interaktion och gestigenkänning[4]. Men om man vill rita upp komplicerad 3D-grafik eller göra sitt helt egna grafiska gränssnitt brukar grafikmotorn XNA användas. Detta kallar Microsoft för *Core Layer Surface Applications*[6].

Då kan man inte använda sig av det inbyggda interaktionssystemet utan måste förlita sig på andra slags modeller. Det finns fyra tekniker man kan använda sig av:

- Man kan analysera den råa datan som skärmen får in. Den använder en synbaserad inputmodell kallad *PixelSense*[7]. Det gör att datan man får är en bild på vad skärmen ser för tillfället[8] som man sedan kan tolka.
- Om man vill skapa interaktiva komponenter som listor och knappar så finns stöd för det via *Core Interaction Framework*[9].
- Det finns stöd för manipuleringsgester som skalning och rotation via *Manipulators*[10].
- Man kan få notifikationer när ett finger kommer i kontakt med skärmen, när det flyttar på sig och när det försvinner från skärmen[11].

Eftersom implementationen i denna rapport ska försöka vara så flexibel som möjlig och inte bunden till någon speciell plattform så kommer endast notifikationerna för fingerhändelser att användas.

2.3 Tidigare arbeten

Det finns redan några ramverk för multi-touch som har fokus på flexibilitet. *Multitouch for Java* (MT4j)[12] är ett exempel. Det är ett open-source-projekt som försöker skapa ett hårdvaruoberoende ramverk för multi-touch för Java. Det använder grafikramverket *processing*[13] för rendering. Ett annat exempel är *PyMT*[14] som är ett liknade ramverk för Python. Det har stöd för Windows 7, MacOSX och Linux. Skillnaden hos dessa ramverk och det som denna rapport tänker undersöka är att de är bundna till ett grafikramverk.

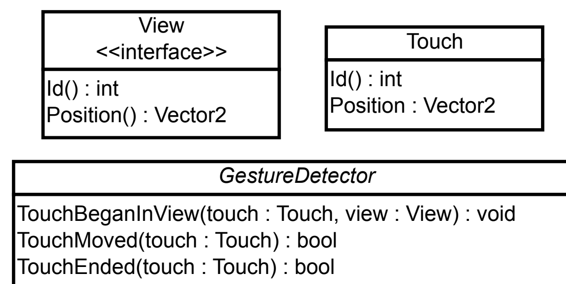
Kapitel 3

Ramverksarkitektur

Följande kapitel beskriver arkitekturen av ramverket. Det följer det objektorienterade paradigmet. Orsaken bakom detta är att implementationsplattformen använder programmeringsspråket `C#` som är ett objektorienterat programmeringsspråk. Det är viktigt här att poängtera att ramverket i rapporten beskrivs oberoende av implementationsspråk. En allmän objektorienterad modell används.

3.1 Överblick

Den centrala delen av ramverket består av interfacen `View`, den abstrakta klassen `GestureDetector` och klassen `Touch`. `View` representerar ett allmänt interaktivt visuellt objekt, `GestureDetector` representerar en allmän gestigenkännare och `Touch` representerar ett finger som rör skärmen. Användare av ramverket låter sina visuella objekt implementera `View`-interfacet. För att kunna känna igen de olika gesterna skapas ett antal subklasser till `GestureDetector`. Dessa klasser är specialiserade på känna igen olika slags gester. Nedan följer ett UML-diagram som beskriver de tre komponenterna.



Figur 3.1. UML-diagram som beskriver de centrala komponenterna i ramverket.

Position på skärmen representeras med en 2-dimensionell vektor `Vector2`.

Arkitekturen av ramverket är väldigt inspirerat av Apples modell *GestureRecognizers*[15]. Den stora skillnaden är att detektorerna i detta ramverk inte är bundna till en speciell `View`, utan kan känna igen gester på ett flertal objekt på samma gång. Detta designbeslut togs för att minimera antalet komponenter systemet måste meddela när en touchhändelse sker. Det behöver bara bli ett meddelande per touchhändelse istället för lika många som det finns objekt på skärmen.

3.2 Komponenter

Nedan följer en beskrivning av de tre komponenterna i ramverket. En anmärkning är att om det inte går att få en unik identifierare för `View` eller `Touch` i den utvecklingsmiljö man använder så fungerar objektreferensen själv som ett bra alternativ. Anledningen till att id används i denna rapport är att det automatiskt ger $O(1)$ i tidskomplexitet för uppslagning i hashtabeller (om id:a är uniformt distribuerade). Se figur 3.3 för information om parametrar och returvärden för de olika metoderna som beskrivs nedan.

View

`View` är ett interface som representerar ett allmänt interaktivt objekt. Detta kommer i fortsättningen att kallas för en *vy* i löpande text. Den har två metoder som måste implementeras.

Metodnamn	Beskrivning
`Id`	Vyns unika identifierare. Används för att associera information till en vy med hjälp av hashtabeller.
`Position`	Vyns position på skärmen. Det spelar ingen roll vart den specificeras i förhållande till innehållet. Kan till exempel vara längst upp till vänster eller i mitten av grafiken.

Touch

`Touch` är en klass som representerar ett tryck med ett finger på skärmen. Detta kommer att i fortsättningen att kallas för en *touch* i löpande text. Beroende på vilken plattform man utvecklar till kan detta objekt se olika ut. I denna rapport så krävs bara att en touch har ett id och en position.

Metodnamn	Beskrivning
`Id`	Touchens unika identifierare. Används för att associera information till en touch med hjälp av hashtabeller.
`Position`	Touchens position på skärmen.

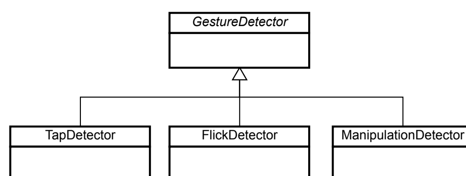
GestureDetector

`GestureDetector` är en abstrakt klass som representerar en allmän gestigenkännare. Den innehåller tre metoder som anropas av ramverkets användare för att meddela att olika typer av touchhändelser har skett. Lägg märke till att det är upp till användaren att identifiera när en touch sätts ner på en vy. Men sedan när touchen rör på sig måste detektorn själv komma ihåg vilken vy som den är sammankopplad med, förslagsvis med en hashtabell där touch-id är nycklar och deras motsvarande vy är värden.

Metodnamn	Beskrivning
`TouchBeganInView`	Anropas av användaren av ramverket när det har upptäckts att ett finger har kommit på skärmen och den är på en vy.
`TouchMoved`	Anropas av användaren av ramverket när en touch rört sig på skärmen, oavsett om det är på en vy eller inte. Gestigenkännaren är sen ansvarig för att avgöra om det är en touch som den bör ta hänsyn till eller inte. Returnerar `true` om den har hanterat touchen, `false` annars.
`TouchEnded`	Anropas av användaren av ramverket när ett finger har lyfts upp från skärmen, oavsett om det var på en vy eller inte. Gestigenkännaren är sen ansvarig för att avgöra om det är en touch som den bör ta hänsyn till eller inte. Returnerar `true` om den har hanterat touchen, `false` annars.

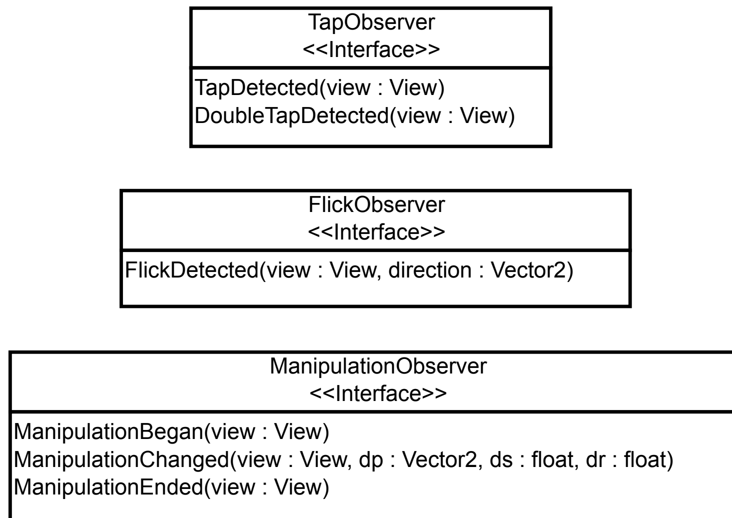
3.3 Gestigenkännare

För att känna igen olika gester skapas subklasser till `GestureDetector`. Varje sådan klass är specialiserad för att känna igen vissa typer av gester. Nya subklasser kan sedan skapas om behovet av nya gester uppkommer. De gester som till en början kommer att stödjas i ramverket är alla som beskrivs under sektion 2.1. Detta möjliggörs med klasserna `TapDetector`, `FlickDetector` och `ManipulationDetector`. `TapDetector` känner igen tap och dubbeltap, `FlickDetector` känner igen flick och `ManipulationDetector` känner igen drag, pinch, spread och rotation. Nedan följer ett UML-diagram som visar hur de hänger ihop.



Figur 3.2. UML-diagram som visar subklasserna till GestureDetector.

När en gest har känts igen måste man kunna notifiera användaren av ramverket på något sätt. Sättet det löses på är att varje detektor paras ihop med ett interface som innehåller ett antal metoder som har med gestens detektion att göra. Detta följer designmönstret *Observer*[16] men tillåter bara en observerare per objekt. Användare av detektorn måste implementera detta interface för att få notifikationer när detektorn detekterar något. När man skapar en detektor skickar man med en observerare med i konsruktorn. `TapDetector`, `FlickDetector` och `ManipulationDetector` har följande observerarinterface:



Figur 3.3. Detektorernas motsvarande observerarinterface.

Några detaljer kan behöva förtydligas kring vissa metoder i dessa interface. Det första är vad `direction` betyder i `FlickDetected`. Det är en normaliserad vektor som är riktad i samma riktning som flicken skedde. Sedan kan parametrarna `dp`, `ds` och `dr` behöva förklaras i `ManipulationChanged`. `dp` står för positionsändring hos vyn. `ds` står för skaländringsfaktor hos vyn. Slutligen står `dr` för rotationsändring hos vyn.

Kapitel 4

Algoritmer

I detta kapitel beskrivs de algoritmer som används för att känna igen gesterna i ramverket. Alla algoritmer beskrivs i löpande text och kodexempel refereras till bilaga A. Orden startposition använt i detta sammanhang och hänvisar till den position ett finger fick på skärmen när det först upptäcktes.

4.1 Tap

För att känna igen en tap måste man hålla kolla på om ett finger har satts ner och sedan tagits bort utan att flytta på sig inom en viss tid. En detalj som är väldigt viktig är vad man menar med "utan att flytta sig". När man trycker ner ett finger på skärmen är det nästan omöjligt att hålla fingret helt stilla. Små rörelser sker utan att man tänker på det. För att ta hänsyn till det krävs det att man tillåter en viss rörelse utan att ett finger räknas som flyttat. Det kan lätt implementeras genom att definiera en konstant som säger hur långt från startpositionen ett finger får flytta sig och fortfarande räknas som stilla. När ett finger en gång har gått förbi denna gräns så kan det inte längre vara en tappning.

För att detektera dubbeltap kan man spara undan senaste tidpunkten som man detekterade en tappning. När man sedan detekterar en tappning igen tittar man hur lång tid det har gått sen senast och om det är under en viss gräns är det en dubbeltappning.

Se bilaga A.2 för implementation i `C#`.

4.2 Flick

En flick sker när man flyttar ett finger snabbt över skärmen och sedan tar upp det tätt inpå. För att känna igen det måste man mäta hastigheten fingret rört sig på skärmen. Man kan ofta anta att fingerregistrering på skärmen sker med jämna intervall. I så fall räcker det med att titta på hur långt fingret har flyttat sig sedan föregående uppdatering för att få ett bra mått på hastigheten. Men om det inte räcker kan man spara tidpunkten för ett antal senaste touchuppdateringarna. När

fingret försvunnit från skärmen kan man titta på dess föregående positionerna på skärmen och räkna ut medelhastigheten. Hur långt tillbaka i tiden man behöver titta är inte självklart, och kan avgöras genom tester. I implementationen till Microsoft Surface 2 räckte det med att spara de två senaste positionerna fingret haft och räkna ut avståndet mellan dem för att få ett bra mått på hastigheten. Om hastigheten är större än ett tröskelvärde, som man får testa sig till, är det en flick.

Se bilaga A.3 för implementation i `C#`.

4.3 Manipulation

Gesterna drag, pinch, spread och rotation kan alla ske samtidigt på ett objekt utan att några fingrar läggs till eller tas bort från skärmen. Det gör att man kan generalisera dessa gester till en *manipulation*-gest. Den gesten innefattar att flytta, ändra storlek och rotera ett objekt.

Modellen som kommer att användas är att när ett finger trycks ner på en vy läggs den till i en lista på touchpunkter som tillhör vyn. När någon touch sedan flyttar sig uppdaterar man ett antal värden och räknar ut skillnaden mot förut. Dessa värden är centrumet av punkterna (medelvärdet av positionerna), medelavståndet till centrumet och medelvinkel från centrum till alla punkter. Skillnaden i centrumposition från föregående uppdatering tolkas som förflyttning. Medelavståndet till centrum delas med det föregående medelavståndet. Då fås en faktor som är ett mått på hur mycket objektet har skalats om. Skillnaden i medelvinkel från centrum tolkas som rotationskillnad på objektet.

Dessa uträkningar sker bara när fingrar flyttar på sig och inte när de läggs till eller tas bort i en vys lista. Detta eftersom tillägg eller borttagning av fingrar kraftigt påverkar de uträknade värdena och resulterar i hackig interaktion.

En anmärkning är hur uträkningen av rotationsskillnad sker. Det finns ett speciellt fall som kräver lite uppmärksamhet. Tänk att man räknar ut vinkeln från centrumet till en punkt under två närliggande uppdateringar. Första gången är vinkeln $359°$, och andra gången är den exempelvis $361°$. Eftersom man ofta begränsar vinkeln till det första varvet under uträkningen blir vinkeln istället $1°$. Detta påverkar medelvärdet av vinklarna väldigt mycket och ett hack kan sker i rotationen av objektet. För att undvika detta kan man sätta en check på att en rotationskillnad inte får vara högre än ett förbestämt litet värde. Om det är större så meddelar man att ingen rotationskillnad har skett. Detta motverkar plötsliga hopp i rotationen.

Se bilaga A.4 för implementation i `C#`.

Kapitel 5

Resultat

Här presenteras resultatet av utvecklingen. Svårigheter på vägen diskuteras även. Den färdiga implementationen av ramverket finns bifogad som bilaga A.

5.1 Implementationssvårigheter

Under största delen av utvecklingen av ramverket fanns det ej tillgång till en riktigt Microsoft Surface 2. Testning fick göras på en vanlig dator och touch-input fick simuleras med programmet *Microsoft Input Simulator*[17]. Detta medförde några mindre komplikationer. Dels för att det inte gick att simulera alla typer av gester som ramverket stödjer och dels för att programmet hängde sig med jämna intervall och omstart krävdes.

När den riktiga enheten väl var på plats framgick att skärmen inte bara reagerade på fingerinput. Det räckte med att bara hålla handen cirka 5 cm över skärmen för att den skulle börja skicka touch-notifikationer. För att komma runt detta filtrerades all input så att bara det som systemet identifierade som fingrar gav utslag.

5.2 Testning

Ett problem med kod som hanterar användarinput är det inte riktigt går att förutspå hur användare kommer att interagera med systemet. Det gör det väldigt svårt att helt säkerställa att allting fungerar som det är tänkt. Detta är extra sant för multi-touch eftersom det är ny typ av interaktion som folk inte riktigt är helt vana vid och alla har olika förväntningar om hur det ska bete sig.

Det kan också vara en dålig idé att låta programmeraren utföra alla tester eftersom den personen vet precis hur koden funkar och omedvetet kan anpassa sig för att interagera på ett säkert sätt.

Med detta i åtanke utfördes ett användartest under utvecklingens gång då personer som inte var inblandade fick interagera med testobjekt på Microsoft Surface 2. Testet gick till så att fem personer samtidigt fick interagera med ett tiotal färgade

kvadrater som gick att flytta, förstora, rotera och trycka på. Sättet som användarna interagerade observerades och låg till grund för eventuella ändringar.

Testet ledde till en konkret ändring i ramverket och det var hur objekten reagerar när man lägger till eller tar bort ett finger under en manipulationsgest. Innan användartestet utfördes så var objektets position alltid centrerad under användarens fingrar när man flyttar/skalar det. Detta ledde till hackigt beteende när ett finger läggs till eller tas bort på ett objekt eftersom centrumpositionen av fingrarna ändras markant då. Den ändringen som gjordes i ramverket var att inte centrera objektet under fingrarna alls utan bara hålla koll på hur fingrarna har flyttat sig under manipulationens gång och uppdatera objektets position efter det.

5.3 Slutversionen

Den slutgiltiga implementationen av ramverket finns bifogad som bilaga A. Alla gester fungerade utan komplikationer vid användartester. Antalet fingrar på skärmen påverkade inte mjukheten i interaktionen. Fem personer med två händer på skärmen var kunde utan problem interagera med flera objekt samtidigt utan att någon försämring i prestanda märktes. Det enda problemet var att Microsoft Surface 2 inte är lika stabil i sin fingerregistrering när så många fingrar (över 50 st) befinner sig på skärmen samtidigt. Stundtals kunde då fingrar tappas bort från skärmen.

Kapitel 6

Diskussion och slutsats

Ramverket som utvecklades mötte målet på att klara över 50 fingrar på samma gång. Men är det flexibelt? Med flexibelt menas att så lite som möjligt tas för givet kring utvecklingsmiljön där ramverket kommer att användas och att det lätt kan utvidgas. Det enda som togs för givet kring utvecklingsmiljön är att det går att få notifikationer när ett ett finger sätt ner, flyttas och tas bort från skärmen. Det är någonting som finns i de vanligaste utvecklingsmiljöerna, t.ex till iPhone[3], Android[18] och Microsoft Surface 2[11]. Dessutom kräver interfacet `View` bara att de visuella objektet har en position på skärmen, vilket varje visuellt objekt borde kunna tilldelas. Detta gör att ramverket kan implementeras på nästan vilken plattform som helst som använder ett objektorienterat programmeringsspråk. Ramverket kan också lätt att utvidga med nya gester genom att skapa nya subklasser till `GestureDetector`. Dessa egenskaper borde göra att ramverket med fördel kan användas t.ex när 3D-rendering med ett externt ramverk behöver användas.

Litteraturförteckning

[1] OpenGL. URL: http://www.opengl.org/, hämtad 2012-04-01.

[2] Microsoft. XNA Developer Center. URL: http://msdn.microsoft.com/en-us/aa937791, hämtad 2012-04-01.

[3] Apple. Event Handling Guide for iOS. URL: http://developer.apple.com/library/ios/#documentation/EventHandling/Conceptual/EventHandlingiPhoneOS/MultitouchEvents/MultitouchEvents.html#/, 2011-03-10, hämtad 2012-04-01.

[4] Microsoft. Developing Presentation Layer Surface Applications. URL: http://msdn.microsoft.com/en-us/library/ff727733.aspx, hämtad 2012-04-01.

[5] Microsoft. Microsoft Surface 2. URL: http://www.microsoft.com/surface/en/us/default.aspx, hämtad 2012-04-04.

[6] Microsoft. Developing Core Layer Surface Applications. URL: http://msdn.microsoft.com/en-us/library/ff727894.aspx, hämtad 2012-04-01.

[7] Microsoft. PixelSense. URL: http://www.microsoft.com/surface/en/us/pixelsense.aspx, hämtad 2012-04-04.

[8] Microsoft. Capturing and Rendering a Raw Image. URL: http://msdn.microsoft.com/en-us/library/ff727887.aspx, hämtad 2012-04-04.

[9] Microsoft. Core Interaction Framework. URL: http://msdn.microsoft.com/en-us/library/ff727877.aspx, hämtad 2012-04-04.

[10] Microsoft. Manipulations and Inertia in the Core Layer. URL: http://msdn.microsoft.com/en-us/library/ff727883.aspx, hämtad 2012-04-05.

[11] Microsoft. TouchTarget Class. URL: http://msdn.microsoft.com/en-us/library/microsoft.surface.core.touchtarget.aspx, hämtad 2012-04-05.

[12] Multitouch for Java. URL: http://www.mt4j.org/mediawiki/index.php/Main_Page, hämtad 2012-04-11.

[13] Processing. URL: http://processing.org/, hämtad 2012-04-11.

[14] PyMT. URL: http://pymt.eu/, hämtad 2012-04-11.

[15] Apple. Gesture Recognizers. URL: http://developer.apple.com/library/ios/#documentation/EventHandling/Conceptual/EventHandlingiPhoneOS/GestureRecognizers/GestureRecognizers.html#/, 2011-03-10, hämtad 2012-04-11.

[16] Microsoft. Exploring the Observer Design Pattern. URL: http://msdn.microsoft.com/en-us/library/ee817669.aspx, hämtad 2012-04-08.

[17] Microsoft. Input Simulator. URL: http://msdn.microsoft.com/en-us/library/ff727737.aspx, hämtad 2012-04-09.

[18] Google. Android Input Events. URL: http://developer.android.com/guide/topics/ui/ui-events.html, hämtad 2012-04-11.

Bilaga A

Källkod

A.1 GestureDetector.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Surface.Core;
using Innovationssafari.Views;

namespace Innovationssafari.Input
{
    public abstract class GestureDetector
    {
        private bool exclusiveTouch;

        protected const float TAP_RADIUS = 15;
        protected const float DOUBLE_TAP_RADIUS = 50;

        public bool ExclusiveTouch
        {
            get { return exclusiveTouch; }
            set { exclusiveTouch = value; }
        }

        public abstract void TouchBeganInView(TouchPoint point, View view);

        public abstract bool TouchMoved(TouchPoint touch);

        public abstract bool TouchEnded(TouchPoint touch);
    }
}
```

A.2 TapDetector.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Innovationssafari.Views;
using Microsoft.Xna.Framework;
```

```
using Microsoft.Surface.Core;
using System.Diagnostics;

namespace Innovationssafari.Input
{
    public interface TapObserver
    {
        void TapOccured(View view);
        void DoubleTapOccured(View view);
    }

    class TapDetector : GestureDetector
    {
        private HashSet<View> touchedViews;
        private Dictionary<View, Vector2> touchDownPositionForView;
        private Dictionary<int, View> viewForTouchId;
        TapObserver tapObserver;
        private Dictionary<int, DateTime> lastTapDateForViewId;
        private Dictionary<int, Vector2> lastTapPosForViewId;
        private Dictionary<int, DateTime> lastTouchDownTimeForViewId;

        public TapDetector(TapObserver tapObserver)
        {
            touchDownPositionForView = new Dictionary<View, Vector2>();
            touchedViews = new HashSet<View>(new View.EqualityComparer());
            viewForTouchId = new Dictionary<int, View>();
            lastTapDateForViewId = new Dictionary<int, DateTime>();
            lastTapPosForViewId = new Dictionary<int, Vector2>();
            lastTouchDownTimeForViewId = new Dictionary<int, DateTime>();
            this.tapObserver = tapObserver;
        }

        public override void TouchBeganInView(TouchPoint touch, View view)
        {
            if (touchedViews.Contains(view)) return;

            touchDownPositionForView.Add(view, new Vector2(touch.X, touch.Y));

            lastTouchDownTimeForViewId[view.ID] = DateTime.Now;

            viewForTouchId.Add(touch.Id, view);
            touchedViews.Add(view);
        }

        public override bool TouchMoved(TouchPoint touch)
        {
            bool hasHandledTouch = false;

            if (viewForTouchId.ContainsKey(touch.Id))
            {
                hasHandledTouch = true;
                View view = viewForTouchId[touch.Id];
                Vector2 touchPos = new Vector2(touch.X, touch.Y);
                if (Vector2.Distance(touchDownPositionForView[view], touchPos) > TAP_RADIUS)
                {
                    touchDownPositionForView.Remove(view);
                    viewForTouchId.Remove(touch.Id);
                    touchedViews.Remove(view);
                }
            }

            return hasHandledTouch;
```

```
        }

        public override bool TouchEnded(TouchPoint touch)
        {
            bool hasHandledTouch = false;

            if (viewForTouchId.ContainsKey(touch.Id))
            {
                hasHandledTouch = true;
                View view = viewForTouchId[touch.Id];


                touchDownPositionForView.Remove(view);
                viewForTouchId.Remove(touch.Id);
                touchedViews.Remove(view);


                if ((DateTime.Now - lastTouchDownTimeForViewId[view.ID]).TotalSeconds <= 1.0f)
                {

                    bool doubleTapping = false;

                    if (lastTapPosForViewId.ContainsKey(view.ID) &&
                        lastTapDateForViewId.ContainsKey(view.ID))
                    {
                        bool inTime = (DateTime.Now - lastTapDateForViewId[view.ID]).TotalSeconds <= 1.0f;
                        Vector2 lastTapPos = lastTapPosForViewId[view.ID];
                        Vector2 touchPos = new Vector2(touch.X, touch.Y);
                        if (Vector2.Distance(lastTapPos, touchPos) <= DOUBLE_TAP_RADIUS && inTime)
                        {
                            doubleTapping = true;
                        }
                    }

                    if (!doubleTapping)
                    {
                        tapObserver.TapOccured(view);

                        lastTapDateForViewId[view.ID] = DateTime.Now;
                        lastTapPosForViewId[view.ID] = new Vector2(touch.X, touch.Y);
                    }
                    else
                    {
                        tapObserver.DoubleTapOccured(view);

                        lastTapPosForViewId.Remove(view.ID);
                        lastTapDateForViewId.Remove(view.ID);
                    }
                }

            }
            return hasHandledTouch;
        }
    }
}
```

A.3 FlickDetector.cs

```
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Text;
using Innovationssafari.Views;
using Microsoft.Xna.Framework;
using Microsoft.Surface.Core;

namespace Innovationssafari.Input
{
    public interface FlickObserver
    {
        void FlickDetected(View view, Vector2 direction);
    }

    class FlickDetector : GestureDetector
    {
        private const float FLICK_MIN_SPEED = 7;

        private Dictionary<int, Vector2[]> lastPositionsForTouchId;
        private Dictionary<int, View> viewForTouchId;
        FlickObserver flickObserver;


        public FlickDetector(FlickObserver flickObserver)
        {
            lastPositionsForTouchId = new Dictionary<int, Vector2[]>();
            viewForTouchId = new Dictionary<int, View>();
            this.flickObserver = flickObserver;
        }

        public override void TouchBeganInView(TouchPoint touch, View view)
        {
            Vector2 touchPosition = new Vector2(touch.X, touch.Y);
            viewForTouchId.Add(touch.Id, view);
            lastPositionsForTouchId[touch.Id] = new Vector2[2];
            lastPositionsForTouchId[touch.Id][0] = touchPosition;
            lastPositionsForTouchId[touch.Id][1] = touchPosition;
        }

        public override bool TouchMoved(TouchPoint touch)
        {
            bool hasHandledTouch = false;

            if (viewForTouchId.ContainsKey(touch.Id))
            {
                lastPositionsForTouchId[touch.Id][0] = lastPositionsForTouchId[touch.Id][1];
                lastPositionsForTouchId[touch.Id][1] = new Vector2(touch.X, touch.Y);

                hasHandledTouch = true;
            }

            return hasHandledTouch;
        }

        public override bool TouchEnded(TouchPoint touch)
        {
            bool hasHandledTouch = false;

            if (viewForTouchId.ContainsKey(touch.Id))
            {
                Vector2 p1 = lastPositionsForTouchId[touch.Id][1];
                Vector2 p0 = lastPositionsForTouchId[touch.Id][0];
```

```
                if (Vector2.Distance(p1, p0) >= FLICK_MIN_SPEED)
                {
                    Vector2 diff = p1 - p0;
                    diff.Normalize();
                    flickObserver.FlickDetected(viewForTouchId[touch.Id], diff);
                }

                lastPositionsForTouchId.Remove(touch.Id);

                hasHandledTouch = true;
            }

            return hasHandledTouch;
        }
    }
}
```

A.4 ManipulationDetector.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Surface.Core;
using Innovationssafari.Events;
using Innovationssafari.Views;
using Microsoft.Xna.Framework;
using System.Diagnostics;
using Innovationssafari.Utilities;

namespace Innovationssafari.Input
{
    public interface ManipulationObserver
    {
        void ManipulationBegan(View view);
        void ManipulationChanged(View view,
                                 Vector2 positionDelta,
                                 float scaleDelta,
                                 float rotationDelta,
                                 int numberOfFingers);
        void ManipulationEnded(View view);
    }

    public class PinchDetector : GestureDetector
    {
        Dictionary<int, View> viewForTouchId;
        Dictionary<View, Dictionary<int, Vector2>> touchPositionsForView;
        HashSet<View> movedViews;
        Dictionary<View, Vector2> touchDownPositionForView;
        ManipulationObserver manipulationObserver;
        public int MaxTouchesPerObject;

        public PinchDetector(ManipulationObserver manipulationObserver)
        {
            this.manipulationObserver = manipulationObserver;

            MaxTouchesPerObject = 0;

            touchPositionsForView = new Dictionary<View, Dictionary<int, Vector2>>();
            viewForTouchId = new Dictionary<int, View>();
```

```
    movedViews = new HashSet<View>(new View.EqualityComparer());
    touchDownPositionForView = new Dictionary<View, Vector2>(new View.EqualityComparer());

}

public override void TouchBeganInView(TouchPoint touch, View view)
{

    bool pinchIsAlreadyPresent = touchPositionsForView.ContainsKey(view);
    if(!pinchIsAlreadyPresent) {
        touchPositionsForView[view] = new Dictionary<int, Vector2>();
    }


    Dictionary<int, Vector2> touchPositions = touchPositionsForView[view];


    if (MaxTouchesPerObject != 0 && touchPositions.Count >= MaxTouchesPerObject) return;

    touchPositions[touch.Id] = new Vector2(touch.X, touch.Y);

    viewForTouchId[touch.Id] = view;


    if (!pinchIsAlreadyPresent)
    {
        touchDownPositionForView[view] = touchPositions[touch.Id];
        manipulationObserver.ManipulationBegan(view);
    }
    else
    {
        manipulationObserver.ManipulationChanged(view, Vector2.Zero, 1, 0, touchPositions.Count);
    }
}

public override bool TouchMoved(TouchPoint touch)
{
    bool hasHandledTouch = false;
    if (viewForTouchId.ContainsKey(touch.Id))
    {
        hasHandledTouch = true;

        View view = viewForTouchId[touch.Id];
        Dictionary<int, Vector2> touchPositions = touchPositionsForView[view];

        ICollection<Vector2> lastTouchVectors = touchPositions.Values;

        Vector2 lastCenter = centerOfTouchPositions(lastTouchVectors);
        float lastMeanCenterDistance = meanCenterDistanceToPoint(lastTouchVectors, lastCenter);
        float lastMeanAngle = meanAngleFromPoint(lastTouchVectors, lastCenter);

        touchPositions[touch.Id] =  new Vector2(touch.X, touch.Y);


        ICollection<Vector2> touchVectors = touchPositions.Values;

        Vector2 center = centerOfTouchPositions(touchVectors);
        float meanCenterDistance = meanCenterDistanceToPoint(touchVectors, center);
        float meanAngle = meanAngleFromPoint(touchVectors, center);

        if (touchPositions.Count == 1)
```

```
            {
                if (!movedViews.Contains(view))
                {
                    if (Vector2.Distance(touchDownPositionForView[view], center) > TAP_RADIUS)
                    {
                        movedViews.Add(view);
                        manipulationObserver.ManipulationChanged(view,
                                                                 center - lastCenter,
                                                                 1,
                                                                 0,
                                                                 touchPositions.Count);
                    }
                }
                else
                {
                    manipulationObserver.ManipulationChanged(view,
                                                             center - lastCenter,
                                                             1,
                                                             0,
                                                             touchPositions.Count);
                }
            }
            else
            {
                float scaleDelta = meanCenterDistance / lastMeanCenterDistance;
                float angleDelta = meanAngle - lastMeanAngle;

                // Avoid sudden rotations
                if (Math.Abs(angleDelta) > 0.2f) angleDelta = 0;

                manipulationObserver.ManipulationChanged(view,
                                                         center - lastCenter,
                                                         scaleDelta,
                                                         angleDelta,
                                                         touchPositions.Count);
            }
        }

        return hasHandledTouch;
    }

    public override bool TouchEnded(TouchPoint touch)
    {
        bool hasHandledTouch = false;
        if (viewForTouchId.ContainsKey(touch.Id))
        {
            hasHandledTouch = true;


            View view = viewForTouchId[touch.Id];
            Dictionary<int, Vector2> touchPositions = touchPositionsForView[view];

            // Remove touch info
            touchPositions.Remove(touch.Id);
            viewForTouchId.Remove(touch.Id);

            if (touchPositions.Count > 0)
            {
                ICollection<Vector2> touchVectors = touchPositions.Values;
                Vector2 center = centerOfTouchPositions(touchVectors);
                float meanCenterDistance = meanCenterDistanceToPoint(touchVectors, center);
                float meanAngle = meanAngleFromPoint(touchVectors, center);
```

```
                manipulationObserver.ManipulationChanged(view, Vector2.Zero, 1, 0, touchPositions.Count);
            }
            else
            {
                // Manipulation Ended

                // Remove view info
                touchDownPositionForView.Remove(view);
                if (movedViews.Contains(view)) movedViews.Remove(view);
                touchPositionsForView.Remove(view);

                manipulationObserver.ManipulationEnded(view);

            }
        }

        return hasHandledTouch;
    }

    private Vector2 centerOfTouchPositions(ICollection<Vector2> vectors)
    {
        Vector2 center = Vector2.Zero;
        foreach (Vector2 vector in vectors) center += vector;
        center /= vectors.Count;
        return center;
    }

    private float meanCenterDistanceToPoint(ICollection<Vector2> vectors, Vector2 point)
    {
        float meanDistance = 0;
        foreach (Vector2 vector in vectors) meanDistance += Vector2.Distance(vector, point);
        meanDistance /= vectors.Count;
        return meanDistance;
    }

    private float meanAngleFromPoint(ICollection<Vector2> vectors, Vector2 source)
    {
        float meanAngle = 0;
        foreach (Vector2 vector in vectors)
        {
            meanAngle += VectorHelper.DirectionToPoint(source, vector);
        }
        meanAngle /= vectors.Count;
        return meanAngle;
    }

  }
}
```